IPSJ SIG Technical Report

Vol.2011-ARC-197 No.1
Vol.2011-HPC-132 No.1
2011/11/28

A Preliminary Evaluation of Chapel with Molecular Dynamics Simulation

Nan Dun[†1] and Kenjiro Taura[†1]

This paper presents a preliminary study of the programmability and performance of the high-level parallel language Chapel in an implementation of molecular dynamics simulation. Our experience show that Chapel has good expressiveness to describe parallelism, which allows users to effectively write parallel program. The evaluation of language features and MD programs provides the performance implication and usage considerations of using Chapel.

1. Introduction

Through the advance of parallel computing, users now have the chance of using powerful computing resources with *massive parallelism* to solve real-world problems. However, the *programmability* of conventional programming models/languages prevents most users who are familiar with programming sequential machines to efficiently and effectively programming modern parallel machines. This is mainly because the lack of high-level abstractions to express parallelism, locks, synchronization, data distributions and communications in mainstream programming languages[1)]. Besides parallel programmability, another challenge comes from the convergence of heterogeneous/hybrid computing with CPU and GPGPU accelerators, which introduces more parallelism, higher performance, and better power efficiency. Thus without a unified high-level programming model, it is difficult to efficiently integrate various low-level programming models and to orchestra data locality among different platforms.

Among many attempts to narrow this gap, redesigning programming languages that inherently adapt to parallel machines is a profound and effective one. Fortunately, language developers have made significant contribution by creating new parallel languages with the consideration of above problems, such as Cilk[2)], Intel Threading Building Blocks (TBB)[3)], Unified Parallel C (UPC), Chapel[4)], X10[5)], Fortress[6)], etc. Since these are languages are new and most users still do not have much experience with them (especially comparing to well-understood mainstream languages), an investigation and evaluation of high-level languages is necessary to serve following purposes:

- Study the programmability of high-level programming language in the context of real world scientific applications.
- Understand the performance implications of using high-level language, especially comparing to an equivalent implementation in low-level language.
- Provide suggestions for writing scientific applications in high-level languages with both expressiveness and performance consideration.
- Give feedbacks for designing a unified high-level programming model for both enhanced programmability and integration of hybrid architectures.

This paper presents a reference implementation of Molecular Dynamics (MD) simulation by using Chapel programming language. Though there are other parallel programming languages (see section 5.1), we believe using Chapel is more generic and intuitive for domain-specific researchers to effortlessly write parallel programs[7)]. Our implementation covers a wide variety of well-known MD simulation models. We evaluate our implementation by comparing it with an identical C (i. e., low-level language) implementation, especially to conduct a study from the viewpoints of both programmability and performance.

2. Background

2.1 Chapel Programming Language

Chapel[1),4),8)] is an object-oriented parallel programming language designed to improve programmability for modern High Performance Computing (HPC) systems with significant parallelism. The Chapel project was started by Cray as part of DARPA's High Productivity Computing Systems (HPCS) programme which aims to create computing systems with following important attributes of productivity: *performance*, *programmability*, *portability*, and *robustness*[9)]. Chapel focuses on programmability and provides many language features to achieve this objective.

[†1] The University of Tokyo

2.1.1 Language Overview

Chapel adopts a *global view model* rather than conventional fragmented model (which is derived from SIMD model). The major advantage of global view model is to set programmers free from difficulty and tedious of writing parallel programs in an explicit task-by-task manner.

Chapel supports the abstractions of task parallelism, data parallelism, and nested parallelism. The parallelism is described using an *implicit multithreading* manner, in which independent computations are mapped to a collection of threads. Low-level threads management (e. g., create and join) is implemented by the compiler and runtime system instead of users.

The unit of physical computation resources, i. e., CPU plus memory, is abstracted as *locale* in Chapel. For example, one machine with its local memory represents a locale in a cluster architecture. Locales are used to control the mapping of data and computation to the physical machine.

Instead of directly compiling source code to generate executable binary, Chapel uses a *source-to-source translation* from Chapel source code to C code that links to a parallel library. Source-to-source translation not only allows Chapel to support interoperability to other languages such as Fortran or CUDA, but also provides users a chance to understand its underlying execution mechanisms by peeking transformed intermediate C code.

2.1.2 Task and Data Parallelism

The parallelism in Chapel can be constructed by one of following parallel statements: `begin`, `cobegin`, `coforall`, and `forall`.

For task parallelism, the `begin` statement is used with `sync` and `single` synchronization variables to construct concurrent tasks in an unstructured way. The `cobegin` and `coforall` statements are more straightforward for common structured tasks. Following code examples illustrate the difference between these statements.

```
begin procA();
begin procB();
// The order of procA() and procB() is unspecified.

cobegin {
    procA(); // Synchronization is implicitly introduced.
    procB();
} // procA() and procB() finish separately and join here.

coforall i in [1..10] {
    procA();
    procB();
} // Repeat above cobegin block for 10 times.
```

Domain is an important data parallelism concept in Chapel. It is used for define data distributions of arrays (which is a mapping from index to data values). Concurrent tasks that manipulate a collection of data can be constructed using `forall` iterations over a domain or array. For example,

```
var Dom: domain(1) = [1..m];  // Domain
var ArrA, ArrB: [Dom] real;   // Aaray
forall a in Arr do { ... }    // Iterate over an array
ArrB = ArrA;  // Implicit element-by-element copy in parallel
```

2.1.3 Locality Control

Locality is important for parallel programs to achieve good performance. Besides customizing domain distribution, there are two other ways of controlling locality by using `on` clause. As shown in following examples, one is to explicit specify the *locale* where computation should happen, the other is a data-driven approach that keeps computation local with data.

```
// Explicit control of computation location
forall loc in Locales do on loc do procedure();
// Data-driven manner
forall i in Arr.domain do on Arr(i) do procedure(Arr(i));
```

2.2 Molecular Dynamics Simulation

Molecular Dynamics (MD) simulation[10),11)] is a scientific applications that uses numerical approach to study physical substances by simulating the motion of their build blocks: *molecules* and *atoms*. In MD simulation, the movements of molecules obey classical mechanics (i. e., Newton's law of motion) and are dynamically determined by the forces between molecules.

Generally, a MD simulation iterates a computation consisting of following two steps: 1) calculate the force for each molecule in system, 2) advance the

Vol.2011-ARC-197 No.1
Vol.2011-HPC-132 No.1
2011/11/28

positions of the molecules to classical laws of motion. The intermolecular forces are calculated based on the modeling of *force field*. A *potential function* used to define the force field is also called *kernel*. A force field usually has two parts of interactions: 1) short range forces (e. g., van der Waals force) that fall off rapidly with distance and thus are typically evaluated only for nearby molecules, and 2) long range forces (i. e., electrostatic forces) that fall off slowly with distance. Therefore, a $O(N^2)$ computation is required for calculating forces (esp. long-range forces) between all pairs of molecules. Long range forces are typically computed using faster approximate approaches. Some of widely used methods includes Ewald summation[12] using Fast Fourier Transform (FFT), and the Fast Multipole Method (FMM)[13],[14].

FMM is a state-of-the-art algorithm for MD simulation because it significantly reduces the computation complexity of long-range force calculation from $O(N^2)$ to $O(N)$. FMM achieves this by using multipole expansions to approximate the effects of groups of molecules instead of individual molecules with specified accuracy. The groups are further organized in a hierarchical way when the distance between them increases. Due to the lack of space, we refer interested users to further readings[11],[13],[14]. FMM has several variants, such as Kernel-Independent FMM (KIFMM)[15] that avoids to expand underlying kernel. In order to harness the power of parallel machines, recent research effort has also been dedicated to the parallelization of FMM[16],[17]. One of major objectives of our work is also to parallelized FMM, however, by using a language approach rather than other algorithmic approach.

3. Molecular Dynamics in Chapel

We implemented MD programs in Chapel based on an existing collection of C implementation[11]. By this method, we are able to investigate the easiness of rewriting a serial program to a parallel one by Chapel. Additionally, this approach allows us to compare the performance with the performance of a well-understood conventional language. In this paper, we extensively use FMM instead of other programs as the example for description.

3.1 Data Structures

The basic data structure is the representation of molecule. The molecular structure is implemented by `record` type that holds three `vectors` and other attributes, as shown below. We choose `record` rather than `tuple` mainly because of performance considerations (see section 4.2.2).

```
record vector { var x, y, z: real; };
record mol {
  var r, rv, ra: vector; // Position, velocity, acceleration
  var chg: real;         // Electric charge
}
```

Other major data structures include a list to store the cells for multipole expansion, and a list for neighbour molecules to simulate soft-sphere interactions. Both of these lists are implemented using one dimensional array that stores the index of related molecules. A multidimensional array instead of a tree is used to store the expansion coefficients in the hierarchy.

3.2 Potential Functions

For long-range interactions, electrostatic field force that follows the Coulomb's law[11] is used. To prevent molecules with opposite charges from approaching too closely, the pair potential (i. e., Lennard-Jones potential[11]) is used.

3.3 Parallelization

Using Chapel to parallelize the serial code is straightforward and effortless: 1) separate the computation-intensive phase and identify the independent computations encapsulated in `for` loop, then 2) change the `for` into `forall` and protect the globally modified variables with `atomic`[⋆1] statement or `sync` combinations.

In our FMM implementation, several major computation intensive phases are parallelized using only `forall` loops, while `begin` and `cobegin` parallel statements are not used. Most of `forall` parallelism are iterations over the expansion cells. Manipulations on the whole array of molecules, such as initialization and leapfrog steps, are left with serial `for` loop. This is because parallelizing loops with less iterations or inexpensive computations may conversely introduce much more overhead than performance gain. Furthermore, we use user-defined *iterators* to replace the default `range` expression for those nested serial loops to reduce more overhead (see section 4.4).

⋆1 Currently, `atomic` statement is not implemented yet.

Vol.2011-ARC-197 No.1
Vol.2011-HPC-132 No.1
2011/11/28

Because of Chapel's high expressiveness of parallelism, we generally find that less than 10% of the original code is modified/appended when we reconstruct the serial program into parallel program.

4. Evaluation

4.1 Experimental Environments

The experimental environments consist of one cluster of 20 machines. Each machine is equipped with Xeon E5530 2.4GHz 8 cores CPU, 24GB MEM, and an uniform installation of Linux 2.6.26, GCC 4.3.2, and Chapel 1.3.0.

Our Chapel implementation consists of several basic language benchmarks and MD simulation programs, where MD programs is based on an open-source C implementation[⋆1] with detailed illustrations[11]. Therefore, there is no algorithmic but only language descriptive differences between these two implementations.

The used compilation options are shown below. Note that `-O3` option is specified because the same level of optimization is used (which can be shown by `--print-commands` option) during the Chapel compilation from intermediate C code to executable.

```
$ chpl prog.chpl -o prog --fast // Chapel compilation
$ gcc prog.c -o prog -O3 -lm    // C compilation
```

To investigate performance bottlenecks, we extensively uses source-to-source compilation feature (i. e., `--codegen` and `--savec` options) by exploring intermediate C code in following experiments.

For each experiment, if not specified, the shown results are the average of 5 identical runs and their standard deviations are considerable small.

4.2 Language Primitives

4.2.1 Arithmetic and Array Indexing

We first investigate the baseline performance of float point arithmetics for both one single variable and an array. Figure 1 shows the performance of conducting 10^6 operations, with the comparison of a direct C implementation. While the plain float point arithmetics perform the same as direct C implementation, the array reference introduce an average of 15% overhead.

⋆1 Online available at `http://www.ph.biu.ac.il/~rapaport/mdbook/index.html`.

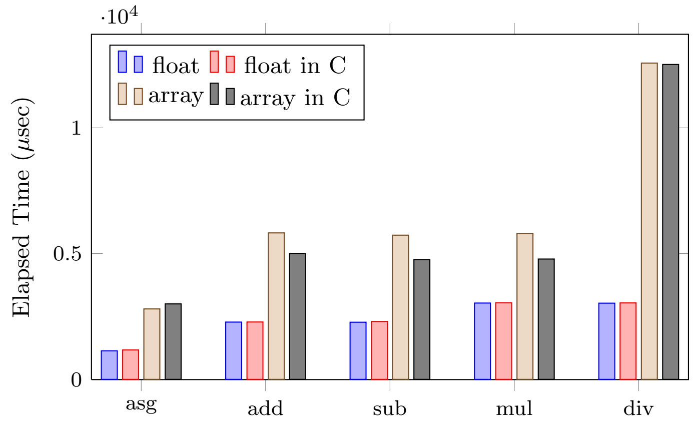

Fig. 1 Comparison of float point arithmetics

4.2.2 Tuple vs. Record

Tuple and *Record* are two light-weight data types for encapsulating a group of data. When translated to intermediate C code, tuple is transformed to multiple dimensional array and record is transformed to the `struct` type. Following code illustrates the correspondence in translation of `tuple` type, `record` type, and their nested constructions.

```
/* Chapel source */                /* C mapping */
var tup: (int, int);               int tup[2];
var nstTup: (tup, tup);            int nstTup[2][2];

record Rec {var x, y: int;}        struct rec {int x, y;}
record nstRec {                    struct nstRec {
    var x, y: Rec; }                   struct rec x, y; }
```

As described in section 3.1, both `tuple` and `record` can be used to implement vector. Figure 2 and 3 show the manipulation performance of 1D and 2D vectors, respectively. 2D vector is implemented by nested types. The number of vectors is 10^6 and we also compare them with a *direct C implementation* using array and `struct`. Results show that using `tuple` has a potential indexing overhead (up to 50%) than using `record`, and the increment of overhead by using nested

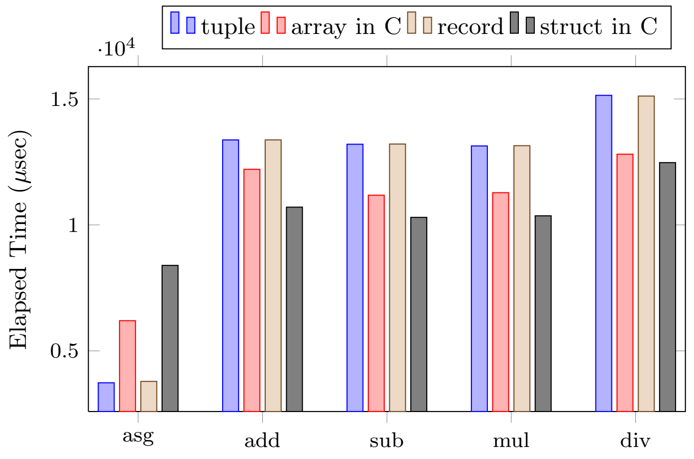

Fig. 2 Performance of manipulations on 1D-vectors

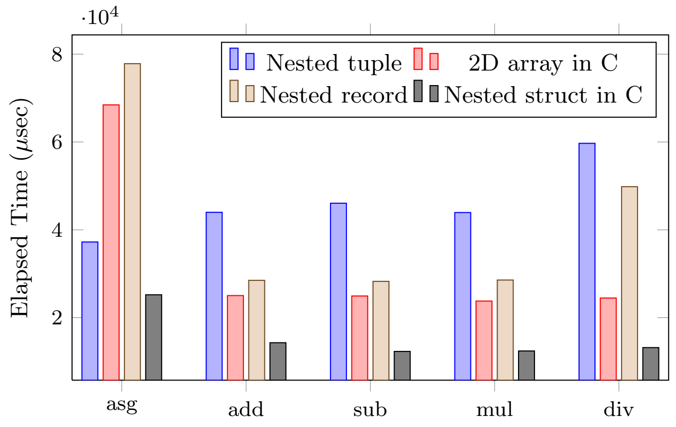

Fig. 3 Performance of manipulation on 2D-vectors

types is much higher comparing to direct C implementation.

4.3 Domain Indexing

In Chapel, domain is classified as *rectangular domain* and *irregular/associative domain*[8]. Rectangular domain describes multidimensional rectangular index

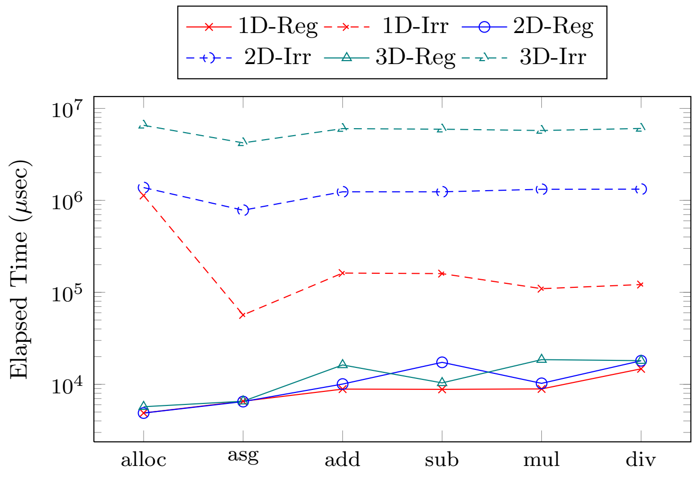

Fig. 4 Indexing performance of arrays with different domains

sets, and irregular domain is like dictionary-style array which can use arbitrary type as index.

To study the performance of domain reference, we compare the throughput of manipulation on arrays that are defined by rectangular domain and associate domain. The size of all arrays is set to 10^6, and the length of a n-dimensional array is $10^{6/n}$. Figure 4 presents the experimental results. Generally, using regular domain is much more efficient (hundreds times faster) than using associate domain because regular domain typically requires $O(1)$ space[8].

4.4 Nested For Loop

The nested loop is common in scientific calculation, and it can be constructed using a *nested iteration* or *zipper iteration* in Chapel. But when the inner loop depends on the outer loop, the iteration can only use the nested way because the *range literal* is evaluated at once before iterations (see example below).

```
// Nested iteration
for i in [1..I] do  // Inner loop depends on outer loop
  for j in [1..I-1] { .. }

// Zipper iteration
```

```
for (i, j) in [1..I, 1..J] do { ... }     // OK
for (i, j) in [1..I, 1..I-1] do { ... }   // NG
```

Figure 5 shows the elapsed time of conducing 10^6 times of accumulation by using different `for`/`while` constructions. Here, "`for-for`" stands for a nested iteration and "`for2`" stands for a zipper iteration. For a n-level nested loops, each level is iterated for $10^{6/n}$ times.

It is clear that the overhead is non-trivial when `for` exists in inner loops. Following intermediate C code shows that a `for` loop in Chapel is translated into a `while` loop surrounded by a pair of domain constructor and destructor procedures which are also iterated by outer loops.

```
// Transformed C code of the for loop
chpl__buildDomainExpr2(&loop_domain, ...);
while (loop_domain) { ... }
chpl__autoDestroy2(loop_variable, ...);
```

Thus, there are three ways to overcome this problem by preventing compiler from inserting the domain construction procedures.

- Define an *iterator* by using `iter` function[8], which preserves the semantics of data parallelism in the `forall` loop[⋆1].
- Use the `while` statement for inner loop, if the inner loop does not need to be executed in parallel.
- Use zipper iteration, if inner loop is independent of outer loop.

4.5 Molecular Dynamics Applications

4.5.1 Serial Execution Performance

Figure 6 shows the performance of the serial version of FMM. Similar as evaluation results in previous sections, a Chapel program generally achieves about 50% of the performance of an identical C program. Though not shown here, other simpler MD programs with fewer array reference can achieve about 60-70% performance of the C implementation.

4.5.2 Parallel Execution Performance

To parallelize a serial program, parallel statements and synchronization are inserted. Figure 7 shows the performance breakdown of FMM phases by a serial version and a parallelized version (but executed serially) FMM programs. For the most computation intensive part (i. e., `multipoleCalc` phase), the parallelization can introduces 5 times of overhead because lock is used in a heavy loop part.

⋆1 However, the parallel iterator is not available now. It will be supported in the future[8].

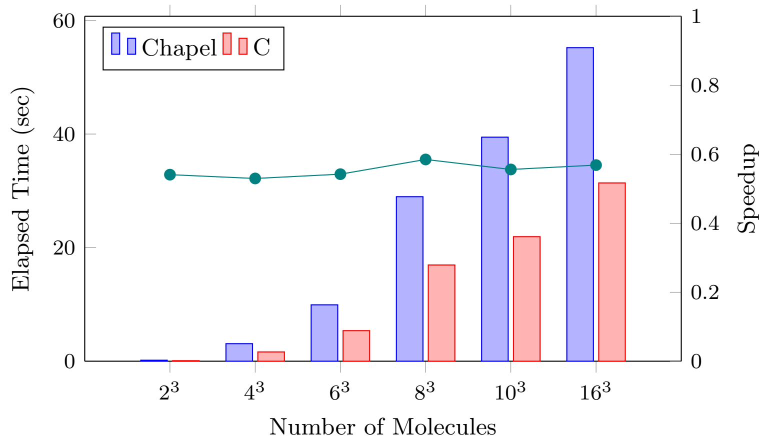

Fig. 6 Performance of serial FMM

Figure 8 shows the scalability of parallel FMM for different number of threads and problem sizes. Figure 9 illustrates the scalability of each phase for $N = 32^3$. For a small problem size (e. g., $N = 8^3$), the performance drops down when the number of threads exceed 4 because there are less calculations for long range interactions and the computation for short range is dominant. When there are more molecules, which suggests larger space, the performance scales up to 8 threads. However, the speedup only achieves 4 for 8 threads, this is because an lock existing in `multipoleCalc` phase leads to significant overhead. We are currently developing a new algorithm for this phase to remove the usage of lock.

4.6 Source Lines of Code

Figure 10 shows the head-to-head comparison of Lines of Code (LOC) between serial programs of our Chapel and original C implementation. Using Chapel saves 20-40% effort to develop a program. Note that the parallel version of programs in Chapel only introduce a small fraction of additional code. For example, our

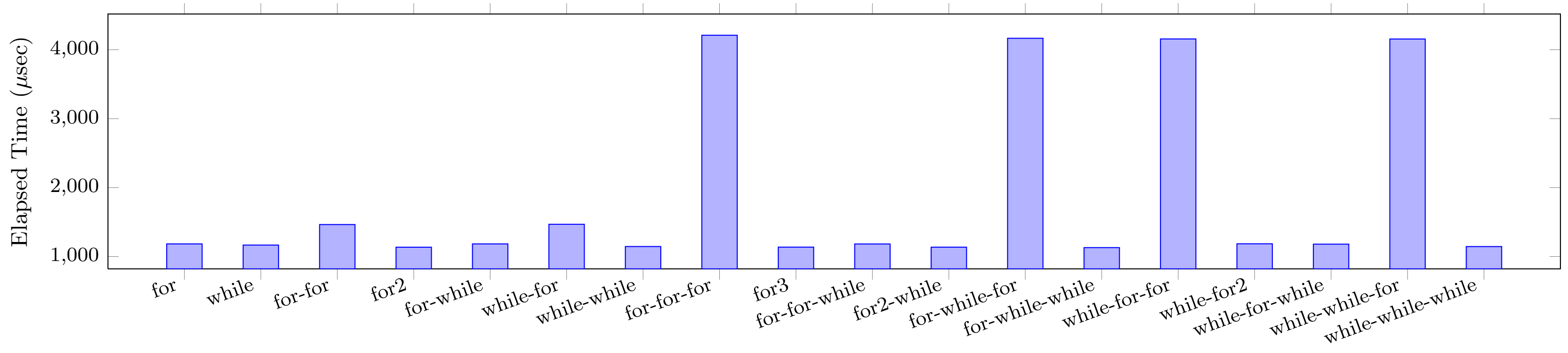

Fig. 5 Performance comparison of traversing various nested loops

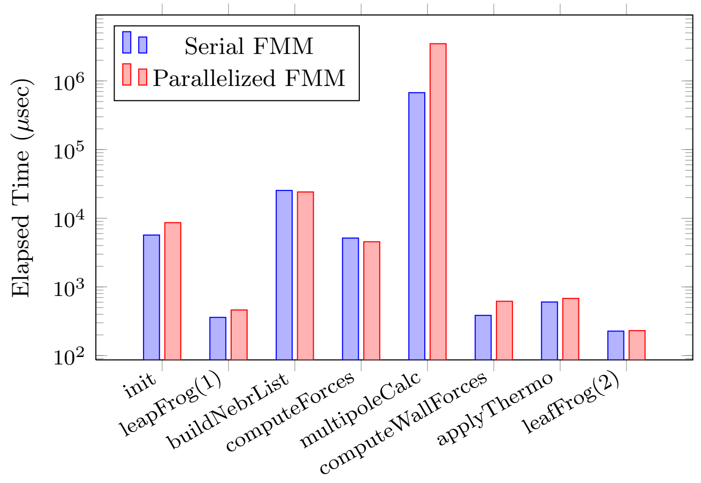

Fig. 7 Serial performance breakdown of serial FMM and parallelized FMM

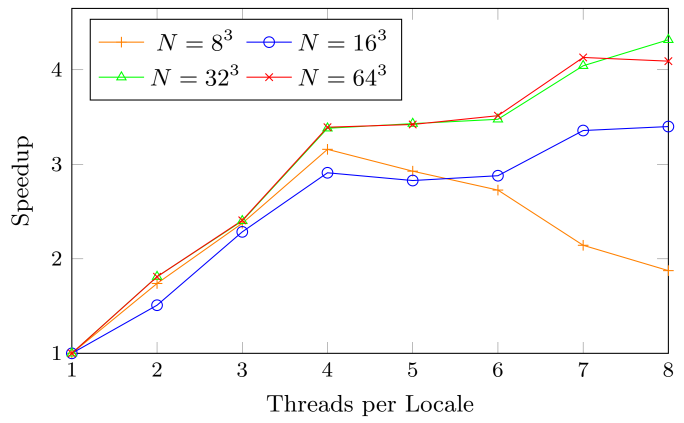

Fig. 8 Scalability of parallelized FMM

parallelized FMM program has only 3% more lines of code than the serial version, which demonstrates that expressive of describing parallelism by Chapel.

5. Related Work

5.1 Parallel Programming Languages

Conventional parallel programming languages (and libraries) includes SPMD languages such as MPI[18], UPC[19], OpenMP[20], HPF[21], Cilk[2], TBB[3].

Besides Chapel, there are two other high-level parallel programming languages

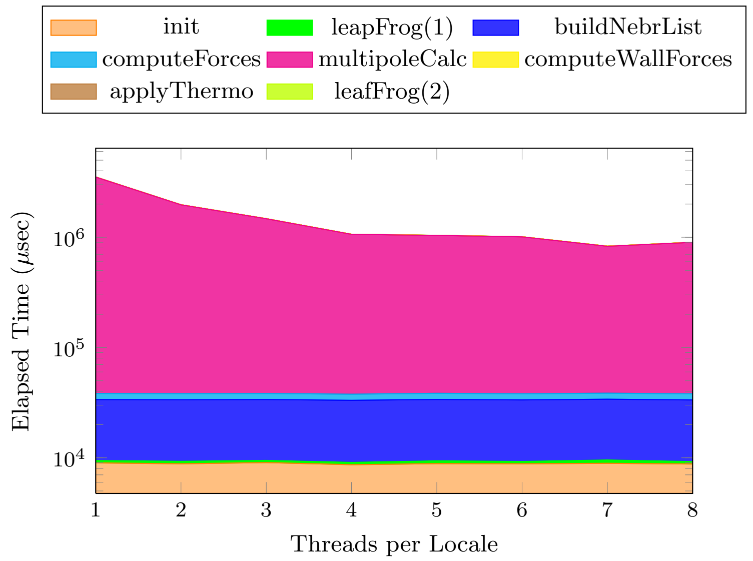

Fig. 9 Parallel performance breakdown of parallelized FMM

that were also initiated by HPCS programme: Oracle's Fortress[6] and IBM's X10[5]. Though these three languages share a common approach of using Partitioned Global Address Space (PGAS) model with multi-threading to adapt various parallel systems, they differ from each other in other aspects[22]. For example, Chapel is an object-oriented programming language aiming for broader community. Fortress is designed to use mathematical syntax to benefit scientific computing users, and X10 targets Java programmers. Though our implementation does use X10 and Fortress, a similar study of the programmability of HPCS languages conclude that they are also expressive for scientific problems[7].

5.2 Practices of Molecular Dynamics Simulations

Prevalent MD simulations are conducted by software approaches on general-purpose computers. Some of widely used software packages include CHARMM[23],

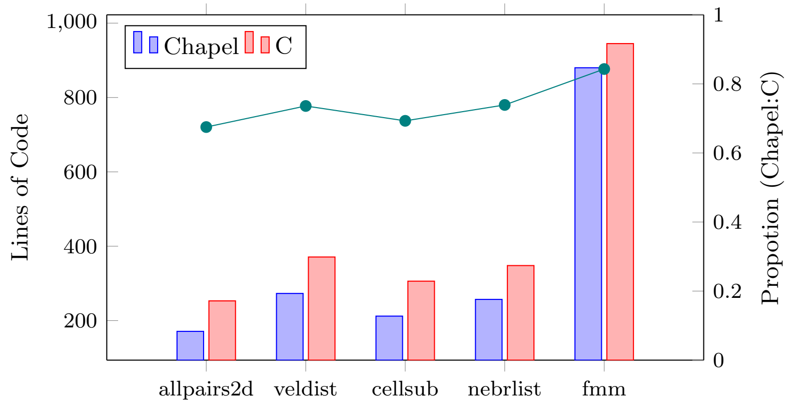

Fig. 10 Comparison of lines of code of serial MD programs

NAMD[24], Desmond[25], Tremole-X[26], etc.⋆1 While most of MD packages are written in C and Fortran, NAMD is implemented using Charm++[27].

There are also parallel machines specifically designed for MD simulations, such as MDGRAPE[28] and Anton[29]. Different from general-purpose computers, they use specialized Application-Specific Integrated Circuits (ASICs) and network to perform computation, which can achieve speedup by several orders of magnitude than general-purpose machines.

Other research projects, such as Folding@home[30] and Docking@Home[31], harness the power of distributed computing resources (e. g., idle CPU of personal computers) to collaboratively perform simulations of protein folding and other molecular dynamics.

6. Conclusion and Future Work

By implementing MD simulation programs in Chapel, we have studied the programmability and performance implication of using a high-level parallel language. Our experience show that Chapel has good expressiveness to describe

⋆1 Refer to `http://en.wikipedia.org/wiki/Molecular_dynamics` for a list of software for MD simulations.

parallelism, which allows users to effectively write parallel program.

Evaluation presents that Chapel has a reasonable performance for scientific applications. Since Chapel is still under development, users also need to be aware of the performance implications of various data and program structures to avoid potential performance overhead.

Our major future work is to develop and optimize the FMM/KIFMM for various parallel architectures including clusters and supercomputers, as well as to investigate the possibility of adopting existing techniques[32),33)] within the Chapel framework. We also plan to extend current implementation to adapt GPGPU-based hybrid computing system, as suggested by recent effort of integrating GPU architecture by using user-defined distributions[34)].

Finally, the source code and documentation of this work is online available at `http://mdoch.googlecode.com/`.

Acknowledgments This work is partially supported by JST, CREST through its research project: "Highly Productive, High Performance Application Frameworks for Post Petascale Computing."

References

1) Chamberlain, B. L., Callahan, D. and Zima, H. P.: Parallel Programmability and the Chapel Language, *International Journal of High Performance Computing Applications*, Vol.21, pp.291–312 (2007).
2) : The Cilk Project, MIT CSAIL Supertech Research Group (online), available from ⟨http://supertech.csail.mit.edu/cilk/⟩ (accessed 2011-10-28).
3) : Intel Threading Building Blocks, Intel Corporation (online), available from ⟨http://software.intel.com/en-us/articles/intel-tbb/⟩ (accessed 2011-10-28).
4) Inc, C.: Chapel Programming Language, Cray. Inc (online), available from ⟨http://chapel.cray.com/⟩ (accessed 2011-10-28).
5) IBM: X10 Programming Language, IBM (online), available from ⟨http://x10-lang.org/⟩ (accessed 2011-10-28).
6) Oracle: Fortress Programming Language, Fortress (online), available from ⟨http://projectfortress.java.net/⟩ (accessed 2011-10-28).
7) Shet, A.G., Elwasif, W.R., Harrison, R.J. and Bernholdt, D.E.: Programmability of the HPCS Languages: A Case Study with a Quantum Chemistry Kernel, *Proc. of IEEE International Symposium on Parallel and Distributed Processing*, IPDPS '08, Miami, Florida, pp.1–8 (2008).
8) Cray Inc.: *Chapel Language Specification v0.8* (2011).
9) Dongarra, J., Graybill, R., Harrod, W., Lucas, R.F., Lusk, E.L., Luszczek, P., McMahon, J., Snavely, A., Vetter, J.S., Yelick, K.A., Alam, S.R., Campbell, R.L., Carrington, L., Chen, T.-Y., Khalili, O., Meredith, J.S. and Tikir, M.M.: DARPAs HPCS Program: History, Models, Tools, Languages, *Advances in Computers: High Performance Computing*, Vol.72, pp.1–100 (2008).
10) Haile, J. M.: *Molecular Dynamics Simulation: Elementary Methods*, Wiley Professional, 1st edition (1997).
11) Rapaport, D. C.: *The Art of Molecular Dynamics Simulation*, Cambridge University Press, 2nd edition (2004).
12) Ewald, P. P.: Die Berechnung optischer und elektrostatischer Gitterpotentiale, *Annalen der Physik*, Vol.369, No.3, pp.253–287 (1921).
13) Greengard, L. and Rokhlin, V.: A Fast Algorithm for Particle Simulations, *Journal of Computational Physics*, Vol.73, pp.325–348 (1987).
14) Kurzak, J. and Pettitt, B. M.: Fast Multipole Method for Particle Dynamics, *Molecular Simulation*, Vol.32, No.10, pp.775–790 (2006).
15) Ying, L., Biros, G. and Zorin, D.: A Kernel-Independent Adaptive Fast Multipole Algorithm in Two and Three Dimensions, *Journal of Computational Physics*, Vol. 196, pp.591–626 (2004).
16) Lashuk, I., Chandramowlishwaran, A., Langston, H., Nguyen, T.-A., Sampath, R., Shringarpure, A., Vuduc, R., Ying, L., Zorin, D. and Biros, G.: A Massively Parallel Adaptive Fast-Multipole Method on Heterogeneous Architectures, *Proc. of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, New York, NY, USA, ACM, pp.58:1–58:12 (2009).
17) Ying, L., Biros, G., Zorin, D. and Langston, H.: A New Parallel Kernel-Independent Fast Multipole Method, *Proc. of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, New York, NY, USA, ACM, pp.14–30 (2003).
18) : MPI: Message Passing Interface, MPI Forum (online), available from ⟨http://www.mpi-forum.org/⟩ (accessed 2011-10-28).
19) : UPC: Unified Parallel C, UC Berkeley/LBNL (online), available from ⟨http://upc.lbl.gov/⟩ (accessed 2011-10-28).
20) : OpenMP, OpenMP Organization (online), available from ⟨http://openmp.org/⟩ (accessed 2011-10-28).
21) : High Performance Fortran, HPF Forum (online), available from ⟨http://hpff.rice.edu/⟩ (accessed 2011-10-28).
22) Weiland, M.: Chapel, Fortress and X10: Novel Languages for HPC, Technical report, HPCx Consortium (2007).
23) Brooks, B.R., III, C. L.B., Jr., A. D.M., Nilsson, L., Petrella, R.J., Roux, B., Won, Y., Archontis, G., Bartels, C., Boresch, S., Caflisch, A., Caves, L. S.D., Cui, Q., Dinner, A.R., Feig, M., Fischer, S., Gao, J., Hodoscek, M., Im, W., Kuczera, K., Lazaridis, T., Ma, J., Ovchinnikov, V., Paci, E., Pastor, R.W., Post, C.B., Pu, J.Z., Schaefer, M., Tidor, B., Venable, R.M., III, H. L.W., Wu, X., Yang, W., York,

D.M. and Karplus, M.: CHARMM: the biomolecular simulation program, *Journal of Computational Chemistry*, Vol.30, No.10, pp.1545–1614 (2009).

24) Phillips, J.C., Braun, R., Wang, W., Gumbart, J., Tajkhorshid, E., Villa, E., Chipot, C., Skeel, R.D., Kalé, L.V. and Schulten, K.: Scalable Molecular Dynamics with NAMD., *Journal of Computational Chemistry*, pp.1781–1802 (2005).

25) Bowers, K.J., Chow, E., Xu, H., Dror, R.O., Eastwood, M.P., Gregersen, B.A., Klepeis, J.L., Kolossvry, I., Moraes, M.A., Sacerdoti, F.D., Salmon, J.K., Shan, Y., and Shaw, D.E.: Scalable Algorithms for Molecular Dynamics Simulations on Commodity Clusters, *Proc. of the ACM/IEEE Conference on Supercomputing*, SC '06 (2006).

26) : Tremole-X, Institute for Numerical Simulation, the University of Bonn (online), available from ⟨http://www.tremolo-x.com/⟩ (accessed 2011-10-28).

27) : Charm++ Parallel Languages, Parallel Programming Laboratory, University of Illinois (online), available from ⟨http://charm.cs.uiuc.edu/research/charm/⟩ (accessed 2011-10-28).

28) Narumi, T., Kawai, A. and Koishi, T.: An 8.61 Tflop/s Molecular Dynamics Simulation for NaCl with a Special-Purpose Computer: MDM, *Proc. of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01 (2001).

29) Shaw, D.E., Dror, R.O., Salmon, J.K., Grossman, J.P., Mackenzie, K.M., Bank, J.A., Young, C., Deneroff, M.M., Batson, B., Bowers, K.J., Chow, E., Eastwood, M.P., Ierardi, D.J., Klepeis, J.L., Kuskin, J.S., Larson, R.H., Lindorff-Larsen, K., Maragakis, P., Moraes, M.A., Piana, S., Shan, Y. and Towles, B.: Millisecond-scale molecular dynamics simulations on Anton, *Proc. of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pp.39:1–39:11 (2009).

30) : Folding@home Project, Pande Lab Stanford University (online), available from ⟨http://folding.stanford.edu/⟩ (accessed 2011-10-28).

31) : Docking@Home Project, University of Delaware (online), available from ⟨http://http://docking.cis.udel.edu/⟩ (accessed 2011-10-28).

32) Kurzak, J. and Pettitt, B. M.: Massively Parallel Implementation of a Fast Multipole Method for Distributed Memory Machines, *Journal of Parallel and Distributed Computing*, Vol.65, pp.870–881 (2005).

33) Chandramowlishwaran, A., Williams, S., Oliker, L., Lashuk, I., Biros, G., Vuduc, R. and Bernholdt, E.: Optimizing and Tuning the Fast Multipole Method for State-of-the-Art Multicore Architectures, *Proc. of IEEE International Symposium on Parallel and Distributed Processing*, IPDPS '10, Atlanta, Georgia, pp.1–12 (2010).

34) Sidelnik, A., Chamberlain, B.L., Garzaran, M.J. and Padua, D.: Using the High Productivity Language Chapel to Target GPGPU Architectures, Technical report, University of Illinois (2011).